Vermeiden von nicht mehr aufzuholenden Rückständen

David Yanacek

Algorithmen imitieren das Leben

Seit meinen ersten IT-Unterrichtsstunden im College fasziniert mich die Umsetzung von Algorithmen im echten Leben. Wir können viele der Prozesse, die um uns herum ablaufen, mit Algorithmen imitieren. Das fällt mir besonders auf, wenn ich irgendwo anstehen und warten muss, zum Beispiel im Supermarkt, im Stau oder am Flughafen. Wenn man gelangweilt in der Schlange steht, ist das eine ideale Gelegenheit, um über die Theorie hinter Warteschlangen nachzudenken.

Vor über zehn Jahren habe ich einen Tag in einem Logistikzentrum von Amazon gearbeitet. Ein Algorithmus hat bestimmt, welche Artikel ich aus den Regalen nehme, in welche Boxen ich sie lege und wohin ich diese Boxen bringe. Mit so vielen Menschen gleichzeitig zusammenzuarbeiten war für mich so, als wäre ich Bestandteil eines perfekt orchestrierten physischen Mergesort.

In der Warteschlangentheorie sind kurze Warteschlangen relativ uninteressant. Schließlich beschwert sich niemand über kurze Wartezeiten. Nur wenn sich eine Warteschlange aufstaut und die Menschen gewissermaßen schon bis auf die Straße und um die Hausecke herum anstehen müssen, machen sie sich plötzlich über Durchsatz und Priorität Gedanken.

In diesem Artikel geht es um Strategien, mit denen wir bei Amazon aufgestaute Warteschlangen handhaben, also Designansätze, mit denen wir Warteschlangen schnell abbauen und Workloads priorisieren. Noch wichtiger ist aber, wie sich Warteschlangen von vornherein vermeiden lassen. In der ersten Hälfte beschreibe ich Szenarien, die zu Rückständen führen, und in der zweiten Hälfte Ansätze, mit denen Rückstände bei Amazon vermieden oder elegant aufgelöst werden.

Der Januskopf der Warteschlange

Warteschlangen sind nützliche Tools für die Erstellung zuverlässiger asynchroner Systeme. Sie erlauben es Systemen, eine Nachricht aus einem anderen System zu empfangen und bis zur vollständigen Verarbeitung aufzubewahren, selbst bei langen Strom- oder Serverausfällen sowie Problemen mit anderen abhängigen Systemen. Anstatt Nachrichten bei Ausfällen zu verwerfen, werden solange erneute Zustellversuche ausgeführt, bis sie erfolgreich verarbeitet wurden. Letztendlich machen Warteschlangen Systeme zuverlässiger und verfügbarer, auf Kosten gelegentlicher Latenz durch Neuversuche.

Wir bei Amazon erstellen viele asynchrone Systeme, die Warteschlangen nutzen. Einige Systeme verarbeiten oft langwierige Workflows, bei denen physische Gegenstände transportiert werden, z. B. die Abwicklung von über amazon.com aufgegebenen Bestellungen. Andere Systeme koordinieren zeitaufwendige Schritte. So fordert Amazon RDS zum Beispiel EC2-Instances an, wartet auf deren Start und konfiguriert dann automatisch Ihre Datenbanken. Wieder andere Systeme nutzen Batchverarbeitung. Systeme für die Aufnahme von CloudWatch-Metriken und -Protokollen rufen Daten ab, aggregieren und "glätten" sie als Chunks.

Zwar liegen die Vorteile von Warteschlangen für die asynchrone Verarbeitung von Nachrichten auf der Hand, aber ihre Risiken sind etwas subtiler. Im Laufe der Jahre haben wir festgestellt, dass Warteschlangen, die eigentlich für Verbesserungen sorgen sollen, auch das Gegenteil bewirken können. Tatsächlich können sie die Wiederherstellung nach einem Ausfall deutlich in die Länge ziehen.

Stoppt in einem warteschlangenbasierten System die Verarbeitung, können sich weiterhin eingehende Nachrichten zu einem großen Rückstand aufstauen, was die Verarbeitungsdauer erhöht. Aufgaben werden dann zu spät erledigt, um noch nützlich zu sein, was zu genau jenen Verfügbarkeitseinbrüchen führt, die Warteschlangen eigentlich verhindern sollten.

Anders gesagt: Ein warteschlangenbasiertes System hat zwei Arbeitsmodi, es verhält sich bimodal. Gibt es keine Rückstaus in der Warteschlange, ist die Latenz gering und das System arbeitet schnell. Aber ein Ausfall oder unerwartetes Auslastungsmuster führt dazu, dass mehr Nachrichten eingehen, als verarbeitet werden. Das System beginnt zu kippen. Die Latenz zwischen den Endpunkten wird größer und größer und es dauert oft sehr lange, bis der Rückstand aufgearbeitet ist und das System wieder schnell läuft.

Warteschlangenbasierte Systeme

Um warteschlangenbasierte Systeme in diesem Artikel zu beschreiben, werfen wir einen Blick hinter die Kulissen zweier AWS-Services: AWS Lambda, ein Service zur ereignisbasierten Ausführung Ihres Codes, bei dem Sie sich nicht um die zugrunde liegende Infrastruktur kümmern müssen, und AWS IoT Core, ein verwalteter Service, der vernetzten Geräten eine einfache und sichere Interaktion mit Cloud-Anwendungen und anderen Geräten ermöglicht.

Bei AWS Lambda laden Sie den Code Ihrer Funktion hoch und rufen diese auf eine von zwei Arten auf:

- Synchron: Die Ausgabe Ihrer Funktion wird in der HTTP-Antwort wiedergegeben.
- Asynchron: Die HTTP-Antwort wird sofort wiedergegeben und Ausführung sowie Neuversuche Ihrer Funktion laufen im Hintergrund ab.

Lambda stellt sicher, dass Ihre Funktion selbst bei Serverausfällen ausgeführt wird, also ist eine zuverlässige Warteschlange zur Speicherung Ihrer Anforderung erforderlich. Mit einer zuverlässigen Warteschlange kann Ihre Anforderung erneut ausgeführt werden, wenn die Funktion beim ersten Versuch scheitert.

Mit AWS IoT Core können sich Ihre Geräte und Anwendungen verbinden und PubSub-Themen abonnieren. Wenn Geräte oder Anwendungen Nachrichten senden, erhalten Anwendungen mit einer passenden Subscription eine separate Kopie der Nachricht. Ein Großteil des PubSub-Messaging wird asynchron ausgeführt, da die ohnehin eingeschränkten Ressourcen von IoT-Geräten nicht damit belegt werden sollten, auf den Eingang einer Kopie bei allen Geräten, Anwendungen und Systemen mit einer Subscription zu warten. Das ist besonders wichtig, da ein Subscriber-Gerät offline sein kann, wenn andere Geräte abonnierte Nachrichten veröffentlichen. Wenn das Gerät wieder online ist, wird es zunächst mit aktuellen Informationen versorgt und erhält später die abonnierten Nachrichten (für Informationen zur Programmierung der Nachrichtenverwaltung in Ihrem System nach einer Neuverbindung siehe Persistente MQTT-Sitzungen im *AWS IoT-Entwicklerhandbuch*). Hinter den Kulissen laufen dazu verschiedene persistente und asynchrone Prozesse ab.

Warteschlangenbasierte Systeme wie diese sind oft mit einer zuverlässigen Warteschlange implementiert. SQS bietet zuverlässige, skalierbare Schemata für die mindestens einmalige Zustellung von Nachrichten. Deshalb machen Teams von Amazon für Lambda und IoT bei der Erstellung ihrer skalierbaren asynchronen Systeme regelmäßig davon Gebrauch. In warteschlangenbasierten Systemen *erzeugt* eine Komponente Daten, indem Nachrichten in die Warteschlange eingereiht werden. Eine andere Komponente *verbraucht* diese Daten durch regelmäßiges Abfragen neuer Nachrichten, verarbeitet diese und löscht sie nach der Fertigstellung.

Ausfälle in asynchronen Systemen

Wenn in AWS Lambda eine Funktion langsamer als üblich aufgerufen wird (z. B. aufgrund einer Abhängigkeit) oder sie zwischenzeitlich ausfällt, gehen keine Daten verloren und Lambda führt die Funktion erneut aus. Lambda reiht Ihre Aufrufe in eine Warteschlange ein und arbeitet den Rückstand der Funktion auf, wenn sie wieder funktionsfähig ist. Aber wir müssen berücksichtigen, wie lange diese Aufarbeitung bis zur Wiederherstellung des Normalzustands dauert.

Als Beispiel dient ein System, das während der Verarbeitung von Nachrichten eine Stunde lang ausfällt. Unabhängig von dessen Geschwindigkeit und Verarbeitungskapazität ist für eine Stunde nach der Wiederherstellung die doppelte Systemkapazität zur Aufarbeitung erforderlich. In der Praxis kann das System natürlich über mehr als die doppelte Kapazität verfügen, besonders bei elastischen Services wie Lambda, und die Wiederherstellung kann schneller verlaufen. Aber andere Systeme, mit denen Ihre Funktion interagiert, sind möglicherweise nicht darauf ausgelegt, einen derartigen Zuwachs an Verarbeitungsaufwand aufgrund der Aufarbeitung zu handhaben. In diesem Fall dauert die Wiederherstellung noch länger. Asynchrone Services bauen bei Ausfällen Rückstände auf, was zu langen Wiederherstellungszeiten führt. Im Gegensatz dazu verwerfen synchrone Services Anforderungen bei Ausfällen, bieten dafür aber eine schnellere Wiederherstellung.

Im Lauf der Jahre waren wir immer wieder versucht, Latenz bei asynchronen Services als unwichtig zu betrachten. Asynchrone Systeme sind oft auf Zuverlässigkeit ausgelegt oder sollen den direkten Aufrufer vor Latenz schützen. In der Praxis stellte sich jedoch heraus, dass die Verarbeitungsdauer sehr wohl eine Rolle spielt und dass selbst von asynchronen Systemen eine Latenz von unter einer Sekunde oder weniger gefordert wird. Werden zwecks höherer Zuverlässigkeit Warteschlangen eingeführt, wird schnell übersehen, dass sie bei Rückständen zu hoher Verarbeitungslatenz führen. Die Aufarbeitung großer Rückstände ist das unerkannte Risiko bei asynchronen Systemen.

Messen von Verfügbarkeit und Latenz

Die Diskussion um Kompromisse zwischen Latenz und Verfügbarkeit wirft eine interessante Frage auf: Wie werden Zielvorgaben bezüglich Latenz und Verfügbarkeit bei asynchronen Services bemessen und festgelegt? Wenn wir Fehlerquoten aus Sicht des *Produzenten* messen, sehen wir nur einen kleinen Teil des Gesamtbildes. Die Verfügbarkeit des Produzenten verhält sich proportional zur Verfügbarkeit der Warteschlange des verwendeten Systems. Wenn wir also mit SQS entwickeln, stimmen die Verfügbarkeit von Produzent und SQS überein.

Wenn wir die Verfügbarkeit des Systems allerdings aus Sicht des *Verbrauchers* messen, kann sie schlechter erscheinen, als sie eigentlich ist, da bei Fehlern möglicherweise Neuversuche stattfinden und dann erfolgreich sind.

Messungen der Verfügbarkeit lassen sich auch aus Dead-Letter Queues (DLQs) ableiten. Kann eine Nachricht nicht mehr neu zugestellt werden, wird sie verworfen oder in eine DLQ eingereiht. Eine DLQ ist schlicht eine separate Warteschlange, in der nicht verarbeitbare Nachrichten zur späteren Analyse und Fehlerbehebung gespeichert werden. Die Quote an verworfenen oder DLQ-Nachrichten ist ein guter Indikator für die Verfügbarkeit, aber sie lässt Probleme oft erst zu spät erkennen. Zwar sind Alarme für hohe DLQ-Volumen sinnvoll, aber wir können uns bei der Problemerkennung nicht allein auf DLQ-Informationen verlassen, da wir sie zu spät erhalten würden.

Wie steht es um die Latenz? Auch hier entspricht die Latenz aus Sicht des *Produzenten* der Latenz unseres Warteschlangenservices. Deshalb bestimmen wir das Alter von Nachrichten in den Warteschlangen. Fälle, in denen Systeme im Rückstand sind oder häufig Fehler auslösen und Neuversuche verursachen, werden so schnell offenbart. Services wie SQS geben an, zu welchem Zeitpunkt eine Nachricht in die Warteschlange gekommen ist. Mit diesem Zeitstempel kann protokolliert werden, wenn eine Nachricht aus der Warteschlange entfernt wird. Daraus lassen sich Metriken über den Rückstand unserer Systeme gewinnen.

Das Problem mit der Latenz ist jedoch noch etwas komplexer. Rückstände sind immerhin zu erwarten und bei nur einigen Nachrichten auch nicht problematisch. Bei AWS IoT wird beispielsweise damit gerechnet, dass Geräte ihre Verbindung verlieren oder Nachrichten nur langsam verarbeiten können. Das liegt an der geringen Leistung und schlechten Internetverbindung vieler IoT-Geräte. Als Betreiber von AWS IoT Core müssen wir den Unterschied zwischen einem erwarteten kleinen Rückstand aufgrund von Verbindungsabbrüchen oder langsamer Verarbeitung von einem unerwarteten systemweiten Rückstand unterscheiden können.

Bei AWS IoT nutzen wie hierzu eine andere Metrik: *AgeOfFirstAttempt*. Bei diesem Messwert wird *die Zeit, die eine Nachricht in der Warteschlange verbracht hat*, vom *aktuellen Zeitpunkt* abgezogen, aber nur, wenn dies der erste Versuch von AWS IoT war, die Nachricht an das Gerät zu senden. Wenn sich Geräte im Rückstand befinden, verfügen wir so über eine genaue Metrik, die nicht verfälscht wird, wenn Geräte Neuzustellungen versuchen oder Nachrichten in Warteschlangen einreihen. Um die Metrik noch genauer zu machen, verwenden wir eine zweite – *AgeOfFirstSubscriberFirstAttempt*. In einem PubSub-System wie AWS IoT gibt es kein praktisches Limit, wie viele Geräte oder Anwendungen ein Thema abonnieren können. Die Latenz ist allerdings höher, wenn Nachrichten an eine Millionen Geräte anstatt nur an eines gesendet werden. Um eine stabile Metrik zu erhalten, erfassen wir eine Timer-Metrik, wenn eine Nachricht zum ersten Mal an den ersten Subscriber eines Themas gesendet wird. Anhand weiterer Metriken messen wir den Fortschritt des Systems bei der Veröffentlichung der verbleibenden Nachrichten.

Die Metrik AgeOfFirstAttempt dient als Frühwarnung vor systemweiten Problemen, hauptsächlich deshalb, weil sie ignoriert, wenn Geräte ihre Nachrichten absichtlich langsam verarbeiten. Hier sollte erwähnt werden, dass Systeme wie AWS IoT mit vielen weiteren Metriken wie dieser instrumentiert werden. Unter allen verfügbaren latenzbezogenen Metriken hat sich die Unterscheidung zwischen der Latenz bei Erst- und Neuversuchen als die beliebteste Strategie bei Amazon herauskristallisiert.

Die Messung von Latenz und Verfügbarkeit asynchroner Systeme ist eine Herausforderung und auch Debugging kann schwierig sein, da Anforderungen zwischen Servern weitergegeben werden und an Orten außerhalb der Systeme verzögert werden können. Um Nachrichten in verteilten Systemen leichter nachverfolgen zu können, geben wir allen Nachrichten in Warteschlangen *Anforderungs-IDs*. Zusätzlich verwenden wir häufig auch Systeme wie X-Ray.

Rückstände in mehrmandantenfähigen Systemen

Viele asynchrone Systeme sind mehrmandantenfähig, sie können von vielen Kunden gleichzeitig genutzt werden. Das verleiht dem Management von Latenz und Verfügbarkeit eine neue Dimension der Komplexität. Der Vorteil der Mehrmandantenfähigkeit ist, dass sich Betriebskosten durch eine konsolidierte Infrastruktur senken lassen. Zudem können Ressourcen durch kombinierte Workloads deutlich effizienter ausgelastet werden. Jedoch erwarten Kunden dieselbe Performance wie bei eigenen Systemen mit einem Mandanten, mit vorhersehbarer Latenz und hoher Verfügbarkeit, unabhängig von den Workloads anderer Kunden.

AWS-Services legen Aufrufern ihre internen Warteschlangen nicht direkt zur Eingabe von Nachrichten offen. Stattdessen verwenden sie schlanke APIs, um Aufrufer zu authentifizieren und deren Informationen an Nachrichten anzufügen, bevor sie in die Warteschlange eingereiht werden. Der Ablauf ist ähnlich wie bei der bereits beschriebenen Lambda-Architektur: Wird eine Funktion asynchron aufgerufen, sendet Lambda Ihre Nachricht in eine Lambda-Warteschlange und ist sofort wieder verfügbar, anstatt seine internen Warteschlangen direkt offenzulegen.

Diese schlanken APIs ermöglichen uns eine faire Drosselung. Fairness ist in einem mehrmandantenfähigen System wichtig, damit die Workloads der Kunden sich einander nicht beeinträchtigen. Bei AWS sorgen oft ratenbasierte Limits pro Kunde für Fairness, wobei für Auslastungsspitzen eine gewisse Flexibilität eingeräumt wird. In vielen unserer Systeme, zum Beispiel in SQS selbst, erhöhen wir das Limit unserer Kunden parallel zu ihrem Unternehmenswachstum. Die Limits dienen als Schutz vor unerwarteten Lastspitzen und geben uns Zeit, hinter den Kulissen ggf. mehr Ressourcen bereitzustellen.

In gewisser Hinsicht funktioniert Fairness in asynchronen Systemen wie die Drosselung in synchronen Systemen. In asynchronen Systemen halten wir Fairness jedoch für wichtiger, da sich dort schnell große Rückstände aufbauen können.

Stellen Sie sich zum Beispiel ein System vor, in dem nicht genügend Vorkehrungen zum Schutz vor Interferenzen anderer Mandanten vorhanden sind. Wenn nun der Traffic eines Kunden plötzlich ansteigt und nicht gedrosselt wird, kann es zu einem systemweiten Rückstand kommen. Dann dauert es bis zu 30 Minuten, bis ein Mitarbeiter herausfindet, was geschehen ist, und das Problem behebt. In diesen 30 Minuten kann das System auf Seiten des Produzenten gut skalieren und alle Nachrichten werden in Warteschlangen eingereiht. Aber die Menge der aufgestauten Nachrichten übersteigt die Kapazität auf Verbraucherseite um das 10-Fache. Entsprechend würden die Aufarbeitung des Rückstands und die Wiederherstellung 300 Minuten dauern. Selbst kurze Lastspitzen können zu stundenlangen Wiederherstellungen führen und entsprechen lange Ausfälle verursachen.

In der Praxis verfügen Systeme von AWS über zahlreiche Ausgleichsmöglichkeiten, um Beeinträchtigungen durch Rückstände zu minimieren oder zu vermeiden. Eine automatische Skalierung hilft zum Beispiel, Probleme bei steigender Auslastung zu vermindern. Allerdings ist es hilfreich, die Auswirkungen von Warteschlangen isoliert zu betrachten, ohne Ausgleichsmöglichkeiten zu berücksichtigen, denn so lassen sich Systeme entwerfen, die auf mehreren Ebenen zuverlässig sind. Hier einige Designmuster, mit denen sich unserer Erfahrung nach große Warteschlangenrückstände und langwierige Wiederherstellungen vermeiden lassen:

- **Asynchrone Systeme sollten auf jeder Ebene geschützt werden.** Da es in synchronen Systemen seltener zu Rückständen kommt, schützen wir sie durch Vorab-Drosselung und Zugangskontrolle. In asynchronen Systemen muss sich jede Systemkomponente selbst vor Überlastung schützen können und verhindern, dass eine Workload einen unverhältnismäßig großen Teil an Ressourcen belegt. Irgendwie schafft es eine Workload immer durch die Zugangskontrolle, weshalb weitere Schutzmaßnahmen erforderlich sind, um Services vor Überlastung zu bewahren.
- **Mehrere Warteschlangen unterstützen das Traffic-Management.** In gewisser Weise schließen sich einzelne Warteschlangen und Mehrmandantenfähigkeit gegenseitig aus. Wenn Aufgaben einmal in einer gemeinsamen Warteschlange eingereiht sind, lassen sich Workloads nur schwer voneinander trennen.

- **Viele Echtzeitsysteme verfügen über FIFO-artige Warteschlangen, werden aber besser nach LIFO-Prinzip genutzt.** Kunden teilen uns oft mit, dass bei Rückständen ihre neuen Daten zuerst verarbeitet werden sollten. Bei einem Ausfall oder einer Lastspitze angesammelte Daten können verarbeitet werden, wenn die nötige Kapazität frei wird.

Wie Amazon resiliente mehrmandantenfähige asynchrone Systeme erstellt

Es gibt mehrere Designmuster, mit denen wir bei Amazon unsere mehrmandantenfähigen asynchronen Systeme auf veränderliche Workloads vorbereiten. Dabei kommen viele Techniken zusammen, aber Amazon nutzt auch viele verschiedene Systeme, wobei für jedes eigene Anforderungen an Verfügbarkeit und Zuverlässigkeit gelten. In den folgenden Kapiteln beschreibe ich einige Designmuster, die wir entweder selbst verwenden oder die in Systemen von AWS-Kunden zum Einsatz kommen.

Workloads in separate Warteschlangen auftrennen

Bei manchen Systemen geben wir jedem Kunden eine eigene Warteschlange, anstatt eine einzige Warteschlange für alle Kunden zu verwenden. Es ist nicht immer kostengünstig, für jeden Kunden oder jede Workload eine Warteschlange hinzuzufügen, denn die Services müssen Ressourcen für die Abfragen aller Warteschlangen aufwenden Doch bei Systemen mit wenigen Kunden oder benachbarten Systemen kann diese einfache Lösung nützlich sein. Andererseits, wenn ein System über Kunden im zwei- oder dreistelligen Bereich verfügt, kann es mit den separaten Warteschlangen ziemlich kompliziert werden. Zum Beispiel verwendet AWS IoT nicht eine separate Warteschlange für jedes IoT-Gerät im Universum. Die Abfragekosten würden dabei eher ungünstig ausfallen.

Shuffle-sharding

AWS Lambda ist ein Beispiel für ein System, bei dem die Abfrage einer separaten Warteschlange für jeden einzelnen Lambda-Kunden zu teuer wäre. Die Verwendung einer einzigen Warteschlange würde jedoch einige der Probleme bewirken, die in diesem Artikel beschrieben werden. AWS Lambda verwendet nicht eine einzige Warteschlange, sondern stellt eine feste Anzahl an Warteschlangen bereit und teilt jedem Kunden eine kleine Anzahl an Warteschlangen zu. Bevor eine Nachricht in die Warteschlange gesetzt wird, wird überprüft, welche der angepeilten Warteschlangen die wenigstens Nachrichten enthält, und die Nachricht wird dann in diese Warteschlange gesetzt. Wenn die Workload eines Kunden zunimmt, entsteht ein Rückstand bei den zugeordneten Warteschlangen, aber andere Workloads werden automatisch von diesen Warteschlangen abgezogen. Es sind nicht besonders viele Warteschlangen erforderlich, um eine magische Ressourcenisolation aufzubauen. Das ist nur eine der vielen Schutzfunktionen von Lambda, die aber auch bei anderen Amazon-Services verwendet wird.

Überschüssigen Traffic auf eine separate Warteschlange umleiten

Wenn ein Rückstand bei einer Warteschlange entstanden ist, ist es in manchen Fällen zu spät, neue Prioritäten für den Traffic zu vergeben. Aber wenn die Verarbeitung der Nachricht relativ teuer oder zeitaufwendig ist, lohnt es sich vielleicht doch, die Nachrichten an eine separate Überlaufwarteschlange umzuleiten. Bei bestimmten Amazon-Systemen implementiert der Kundenservice eine verteilte Drosselung. Wenn er Nachrichten für einen Kunden aus der Warteschlange entfernt, der eine konfigurierte Rate überschritten hat, setzt er diese überschüssigen Nachrichten in separate Überlaufwarteschlangen und löscht die Nachrichten aus der primären Warteschlange. Das System arbeitet weiterhin an den Nachrichten in der Überlaufwarteschlange, sobald Ressourcen verfügbar sind. Das kommt einer Prioritätswarteschlange sehr nahe. Eine ähnliche Logik wird manchmal auf der Produzentenseite angewandt. Wenn ein System also eine große Anzahl an Anfragen von einer einzigen Workload annimmt, verdrängt diese Workload nicht die anderen Workloads in der Warteschlange des aktiven Pfads.

Alten Traffic auf eine separate Warteschlange umleiten

Nicht nur überschüssiger Traffic, sondern auch alter Traffic kann umgeleitet werden. Wenn wir eine Nachricht aus der Warteschlange entfernen, können wir überprüfen, wie alt sie ist. Wir können das Alter protokollieren und anhand dieser Information entscheiden, ob die Nachricht zu einer Rückstandswarteschlange geleitet werden soll, die wir erst dann abarbeiten, wenn wir bei der aktiven Warteschlange wieder aufgeholt haben. Wenn eine Auslastungsspitze an einem Punkt auftritt, an dem wir große Datenmengen aufnehmen, und sich ein Rückstand bildet, können wir den zusätzlichen Traffic so schnell an eine andere Warteschlange umleiten, wie wir den Traffic aus einer Warteschlange entfernen und wieder in eine andere setzen können. Dadurch werden die Ressourcen der Verbraucher schneller für die Verarbeitung neuer Nachrichten frei, als wenn der Rückstand einfach der Reihe nach abgearbeitet wird. So erreichen wir annähernd eine LIFO-Reihenfolge.

Alte Nachrichten entfernen (Nachrichten-Lebenszeit)

Manche Systeme werden nicht beeinträchtigt, wenn sehr alte Nachrichten entfernt werden. Bestimmte Systeme verarbeiten zum Beispiel Deltas zu Systemen sehr schnell, führen aber auch regelmäßig vollständige Synchronisierungen durch. Wir nennen diese Systeme zur regelmäßigen Synchronisierung oft "Anti-Entropy Sweepers". In diesen Fällen müssen wir den alten, angestauten Traffic nicht umleiten, sondern können ihn einfach entfernen, wenn er vor dem letzten Sweep eingegangen ist.

Threads (und andere Ressourcen) für Workloads einschränken

Bei der Entwicklung unserer asynchronen Systeme – wie auch unserer synchronen Services – verhindern wir, dass eine Workload zu viele Threads beansprucht. Ein Aspekt von AWS IoT, über den wir noch nicht gesprochen haben, ist die Rules Engine. Kunden können AWS IoT so konfigurieren, dass Nachrichten von ihren Geräten an ihren Amazon Elasticsearch-Cluster, ihren Kinesis Stream usw. geleitet werden. Wenn die Latenz für diese Kundenressourcen ansteigt, aber die Rate der eingehenden Nachrichten konstant bleibt, steigt das Ausmaß der Nebenläufigkeit im

System an. Und weil ein System zu einem beliebigen Zeitpunkt nur mit einem begrenzten Ausmaß an Nebenläufigkeit fertig wird, verhindert die Rules Engine, dass eine Workload zu viele Ressourcen mit Nebenläufigkeit verwendet.

Dies wird auch in Littles Gesetz beschrieben. Es besagt, dass die Nebenläufigkeit in einem System gleich dem Produkt aus Ankunftsrate und durchschnittlicher Latenz der Anfragen ist. Wenn ein Server beispielsweise 100 Nachrichten/Sekunde bei durchschnittlich 100 ms verarbeitet, werden im Durchschnitt 10 Threads verwendet. Wenn die Latenz plötzlich auf 10 Sekunden ansteigen würde, würden plötzlich 1 000 Threads verwendet werden (im Durchschnitt, in der Praxis könnten es mehr sein), wodurch der Thread-Pool schnell ausgelastet wäre.

Die Rules Engine verwendet mehrere Techniken, um dies zu verhindern. Sie verwendet blockierungsfreie Ein- und Ausgaben, um eine Thread-Überlastung zu verhindern. Es gibt aber noch weitere Einschränkungen für die Auslastung eines Servers (zum Beispiel Arbeitsspeicher und Dateideskriptoren, wenn der Client die Verbindungen durchwechselt und bei der Abhängigkeit die Zeit überschritten wird). Ein zweiter möglicher Schutz vor der Nebenläufigkeit ist ein Semaphor, der das Ausmaß an Nebenläufigkeit misst und einschränkt, das für eine einzige Workload zu einem beliebigen Zeitpunkt verwendet werden kann. Die Rules Engine verwendet auch eine ratenbasierte ausgleichende Beschränkung. Da sich Workloads im Laufe der Zeit verändern, passt die Rules Engine die Beschränkungen im Laufe der Zeit automatisch an, um diese Veränderungen auszugleichen. Und weil diese Rules Engine warteschlangenbasiert ist, dient sie hinter den Kulissen als Puffer zwischen IoT-Geräten und der automatischen Skalierung von Ressourcen und Schutzbeschränkungen.

Bei verschiedenen Amazon-Services verwenden wir separate Thread-Pools für jede Workload, um zu verhindern, dass eine Workload alle verfügbaren Threads beansprucht. Wir verwenden auch einen *AtomicInteger* für jede Workload, um die jeweils zulässige Nebenläufigkeit zu beschränken, und Ansätze mit ratenbasierter Drosselung, um ratenbasierte Ressourcen zu isolieren.

Backpressure upstream senden

Wenn eine Workload einen unangemessenen Rückstand verursacht, mit dem der Verbraucher nicht Schritt halten kann, beginnen viele unserer Systeme automatisch damit, Workloads vom Produzenten deutlich abzulehnen. Ein Rückstand von einem Tag baut sich bei einer Workload schnell auf. Selbst wenn die Workload isoliert ist, ist es fehleranfällig und teuer, sie abzuarbeiten. Die Implementierung dieses Ansatzes könnte bereits durch die gelegentliche Messung der Warteschlangentiefe einer Workload (wenn sich eine Workload in der eigenen Warteschlange befindet) und Skalierung einer Drosselungsbegrenzung eingehender Daten (umgekehrt) proportional zum Rückstandsausmaß erfolgen.

In Fällen, in denen wir eine SQS-Warteschlange für mehrere Workloads verwenden, wird dieser Ansatz kompliziert. Es gibt zwar eine SQS-API, die die Anzahl der Nachrichten in der Warteschlange ausgibt, aber es gibt keine API, die die Anzahl der Nachrichten in der Warteschlange mit einem bestimmten Attribut ausgeben kann. Wir könnten trotzdem die Warteschlangentiefe messen und einen entsprechenden Backpressure anwenden, doch dies würde einen zu starken Backpressure auf Workloads verursachen, die zufälligerweise der gleichen Warteschlange angehören. Andere Systeme, wie Amazon MQ, können den Rückstand detaillierter anzeigen.

Backpressure eignet sich nicht für alle Systeme von Amazon. Zum Beispiel akzeptieren wir bei Systemen, die Aufträge für amazon.com verarbeiten, eher Aufträge, selbst wenn ein Rückstand entsteht, anstatt das Akzeptieren neuer Aufträge zu verhindern. Doch natürlich werden im

Hintergrund viele verschiedene Prioritäten beachtet, damit die dringendsten Aufträge am schnellsten verarbeitet werden.

Verzögerungswarteschlangen zum Verschieben von Aufgaben verwenden

Wenn Systeme erkennen können, dass der Durchsatz einer bestimmten Workload verringert werden muss, versuchen wir, eine *Rückzugsstrategie* für diese Workload zu verwenden. Zur Implementierung verwenden wir oftmals eine SQS-Funktion, die die Übertragung einer Nachricht auf einen späteren Zeitpunkt verschiebt. Wenn wir eine Nachricht verarbeiten und für später speichern, setzen wir sie manchmal wieder in eine separate *Surge Queue,* aber stellen den Verzögerungsparameter so ein, dass die Nachricht für einige Minuten in der Delay Queue verborgen bleibt. Dadurch erhält das System die Möglichkeit, aktuellere Daten zu verarbeiten.

Übermäßige übertragene Nachrichten vermeiden

Bei bestimmten Warteschlangenservices wie SQS gelten Beschränkungen für die Anzahl der übertragenen Nachrichten, die an den Verbraucher der Warteschlange übermittelt werden können. Das ist nicht die Anzahl der Nachrichten, die sich in der Warteschlange befinden können (wofür es keine praktische Grenze gibt), sondern die Anzahl der Nachrichten, die die Verbraucherflotte gleichzeitig verarbeitet. Diese Zahl kann zu hoch sein, wenn ein System Nachrichten aus der Warteschlange entfernt, aber sie nicht löscht. Zum Beispiel sind bereits Bugs aufgetreten, wo Code beim Verarbeiten einer Nachfrage eine Ausnahme nicht auffängt und die Nachricht nicht löscht. In diesen Fällen befindet sich die Nachricht aus Sicht des SQS für die *VisibilityTimeout* der Nachricht in der Übertragung. Wenn wir unsere Strategie zur Fehlerbehandlung und Überlastung entwickeln, berücksichtigen wir dabei diese Beschränkungen und tendieren dazu, die überschüssigen Nachrichten an eine andere Warteschlange zu leiten, anstatt sie sichtbar zu lassen.

SQS-FIFO-Warteschlangen haben eine ähnliche, aber subtile Beschränkung. Bei SQS FIFO nehmen Systeme Ihre Nachrichten der Reihenfolge nach für eine bestimmte *Nachrichtengruppe* auf, aber Nachrichten von verschiedenen Gruppen werden in einer beliebigen Reihenfolge verarbeitet. Wenn also ein kleiner Rückstand in einer Nachrichtengruppe entsteht, verarbeiten wir die Nachrichten in anderen Gruppen auch weiterhin. Doch SQS FIFO fragt nur die letzten 20 000 unverarbeiteten Nachrichten ab. Wenn sich also mehr als 20 000 unverarbeitete Nachrichten in einer Unterordnung von Nachrichtengruppen befinden, laufen andere Nachrichtengruppen mit neuen Nachrichten ins Leere.

Warteschlangen für unzustellbare Nachrichten verwenden

Unzustellbare Nachrichten können zu einer Systemüberlastung beitragen. Wenn ein System eine Nachricht in die Warteschlange setzt, die nicht verarbeitet werden kann (vielleicht weil sie einen Input-Prüfungs-Edge-Fall auslöst), kann SQS helfen, indem diese Nachrichten automatisch in eine separate Warteschlange mit der Funktion "Dead-Letter Queue" (DLQ) [Warteschlange für unzustellbare Nachrichten] umgelenkt werden. Es wird eine Warnung ausgelöst, wenn sich Nachrichten in dieser Warteschlange befinden, denn dann gibt es einen Bug, den wir beheben müssen. Der Vorteil der DLQ-Funktion ist, dass wir damit die Nachrichten erneut verarbeiten können, nachdem der Bug behoben wurde.

Zusätzlichen Puffer in Abfrage-Threads für Workload sicherstellen

Wenn eine Workload genug Durchsatz bis zu dem Punkt erzeugt, dass die Abfrage-Threads die ganze Zeit beschäftigt sind, sogar bei der gleichmäßigen Auslastung, dann hat das System möglicherweise einen Punkt erreicht, wo es keinen Puffer mehr gibt, um einen rasanten Anstieg beim Traffic auszugleichen. In diesem Zustand führt ein kleiner Anstieg beim eingehenden Traffic zu einem enormen unverarbeiteten Rückstand und damit zu einer größeren Latenz. Wir planen einen zusätzlichen Puffer bei den Abfrage-Threads ein, um solche Anstiege auszugleichen. Eine mögliche Maßnahme ist, die Anzahl der Abfrageversuche nachzuverfolgen, die leere Antworten ergeben. Wenn jeder Abfrageversuch mindestens eine Nachricht ergibt, dann ist die Anzahl der Abfrage-Threads genau richtig oder zu niedrig für den eingehenden Traffic.

Heartbeat für langfristige Nachrichten verwenden

Wenn ein System eine SQS-Nachricht verarbeitet, gibt SQS dem System eine bestimmte Zeit, um die Verarbeitung der Nachricht abzuschließen, bevor von einem Systemabsturz ausgegangen wird und die Nachricht für einen neuen Versuch an einen anderen Verbraucher zugestellt wird. Wenn der Code weiterhin ausgeführt wird und diese Deadline vergessen wird, kann die gleiche Nachricht mehrmals gleichzeitig zugestellt werden. Wenn der erste Prozessor eine Nachricht noch nach der Zeitüberschreitung verarbeitet, nimmt ein zweiter Prozessor sie auf und verarbeitet sie genauso über die Zeitüberschreitung hinweg, dann ein dritter und so weiter. Diese Gefahr von kaskadenartigen Ausfällen ist der Grund dafür, warum wir unsere Nachrichtenverarbeitungslogik implementieren, um die Arbeit anzuhalten, wenn eine Nachricht abläuft, oder um einen Heartbeat auf diese Nachricht anzuwenden, um SQS daran zu erinnern, dass wir immer noch daran arbeiten. Dieses Konzept entspricht Leases bei der Leader-Auswahl.

Dabei handelt es sich um eine schleichende Gefahr, denn wir sehen, dass die Latenz eines Systems wahrscheinlich während einer Überlastung ansteigt, vielleicht durch längere Anfragen bei einer Datenbank oder überlastete Server. Wenn eine Systemlatenz die VisibilityTimeout-Grenze überschreitet, löst ein bereits überlasteter Service im Grunde genommen eine Fork Bomb auf sich selbst aus.

Cross-Host-Debugging planen

Es ist bereits schwierig, Ausfälle in einem verteilten System zu verstehen. Im verwandten Artikel zum Instrumentieren werden einige unserer Ansätze zum Instrumentieren von asynchronen Systemen beschrieben, vom regelmäßigen Aufzeichnen der Warteschlangentiefen bis hin zum Übernehmen der "Trace Ids" und Integration in X-Ray. Oder wenn unsere Systeme einen komplizierten asynchronen Workflow verarbeiten, der weit über eine triviale SQS-Warteschlange hinausgeht, verwenden wir oftmals einen anderen asynchronen Workflow-Service wie Step Functions mit Erkenntnissen zum Workflow und vereinfachtem verteiltem Debugging.

Fazit

Bei asynchronen Systemen wird schnell die Wichtigkeit der Latenz vergessen. Immerhin ist bei asynchronen Systemen damit zu rechnen, dass es gelegentlich länger dauert, denn ihnen ist eine Warteschlange vorgeschaltet, um zuverlässige Neuversuche durchzuführen. Doch Szenarien mit Überlastungen und Ausfällen können gewaltige, unüberwindbare Rückstände erzeugen, die ein Service nicht in absehbarer Zeit abarbeiten kann. Diese Rückstände können von einer Workload oder einem Kunden stammen, der die Warteschlangen mit einer unerwartet hohen Rate verwendet, von Workloads, deren Verarbeitung teurer als erwartet ist, oder von der Latenz oder Ausfällen in einer Abhängigkeit.

Wenn wir ein asynchrones System erstellen, müssen wir uns auf diese Szenarien mit den Rückständen konzentrieren und sie antizipieren. Mit Techniken wie Prioritäten, Umleitung und Backpressure können wir die Rückstände minimieren.

Weitere Lektüre

- Warteschlangentheorie
- Littles Gesetz
- Amdahlsches Gesetz
- Little A Proof for the Queuing Formula: L = λW, Case Western, 1961
- McKenney, Stochastic Fairness Queuing, IBM, 1990
- Nichols und Jacobson, Controlling Queue Delay, PARC, 2011